



Functional Programming

Week 3 – Functions on Trees

René Thiemann James Fox Lukas Hofbauer Christian Sternagel Tobias Niederbrunner

Department of Computer Science

Last Lecture

- data = tree shaped data
- every value, expression, function has a **type**
- type of **lhs** and **rhs** has to be equal in function definition **lhs = rhs**
- **built-in types**: `Int`, `Integer`, `Float`, `Double`, `String`, `Char`, `Bool`
- user defined **datatypes**

```
data TName =  
    CName1 type1_1 ... type1_N1  
    | ...  
    | CNameM typeM_1 ... typeM_NM  
    deriving Show
```

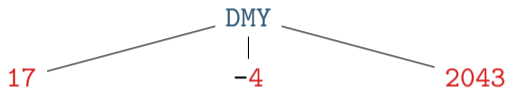
- **constructor** `CNameI :: typeI_1 -> ... -> typeI_NI -> TName`
is a function that is not evaluated
- `TName` is **recursive** if some `typeI_J` is `TName`
- names of types and constructors start with uppercase letters

Examples of Nonrecursive Datatype Definitions

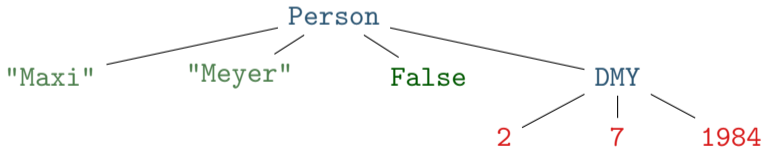
```
data Date = DMY Int Int Integer deriving Show
```

```
data Person = Person String String Bool Date deriving Show
```

- values of type `Date` are trees such as



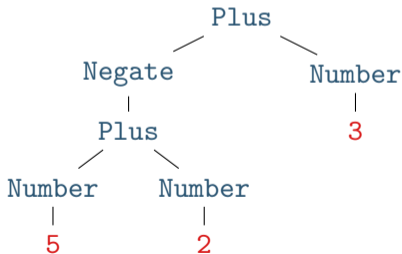
- values of type `Person` are trees such as



Example of Recursive Datatype Definition – Expr

```
data Expr =  
  Number Integer  
  | Plus Expr Expr  
  | Negate Expr  
deriving Show
```

- expression $-(5 + 2) + 3$ in Haskell (as value of type `Expr`):
`Plus (Negate (Plus (Number 5) (Number 2))) (Number 3)`
- expression as tree



Example of Recursive Datatype Definition – Lists

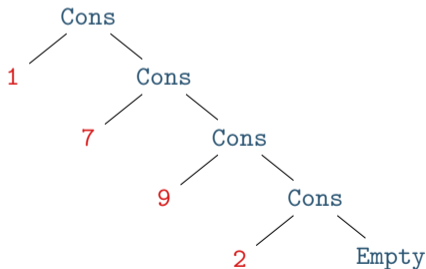
- lists are just a special kind of trees, e.g., lists of `Integers`

```
data List =  
  Empty  
  | Cons Integer List  
deriving Show
```

- example representation of list `[1, 7, 9, 2]`

- in Haskell: `Cons 1 (Cons 7 (Cons 9 (Cons 2 Empty)))`

- as tree:



Function Definitions Revisited

Function Definitions and Expressions

- so far all functions definitions have been of the shape

`funName x1 ... xN = expr`

where

- `x1 ... xN` are variable names;
a function can have arbitrary many parameters (including zero)
- `expr` is an **expression**, i.e., a mathematical expression consisting of
 - variables: `x`, `y`, `xs`, `f`, ...
 - literals: `5`, `3.4`, `'a'`, `"hello"`, ...
 - function applications: `pi`, `square expr`, `average expr1 expr2`, ...
 - constructor applications: `True`, `Number expr`, `Cons expr1 expr2`, ...
 - operator applications: `- expr`, `expr1 + expr2`, ...
 - parenthesis
- remark: function and constructor applications bind stronger than operator applications
 $(\text{square } 2) + 4 = \text{square } 2 + 4 \neq \text{square } (2 + 4)$
- this lecture: **extend shape of function definitions**,
in particular to define functions on tree shaped data

Creating New Values – Expr Example

- creation of new values is easily possible using constructors
- example: consider Expr datatype

```
data Expr = Number Int | Plus Expr Expr | Negate Expr
```

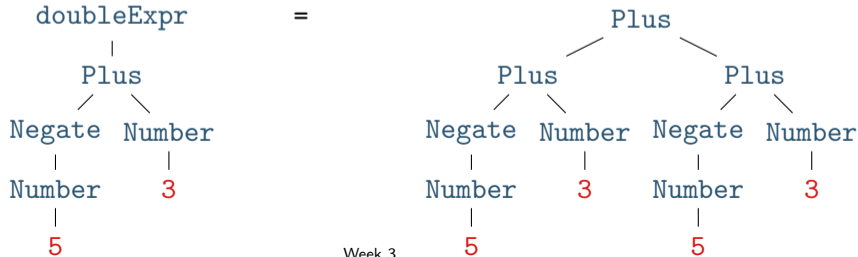
(in the remainder of the lecture “**deriving Show**” is omitted)

- task: define a function for doubling, i.e., multiplication by 2
- solution:

```
doubleNum x = x + x      -- doubling a number
```

```
doubleExpr e = Plus e e  -- doubling an expression
```

- evaluation: `doubleExpr` =



Creating New Values – Person Example

- consider `Person` datatype of last lecture

```
data Date = DMY Int Int Integer
```

```
data Person = Person String String Bool Date
```

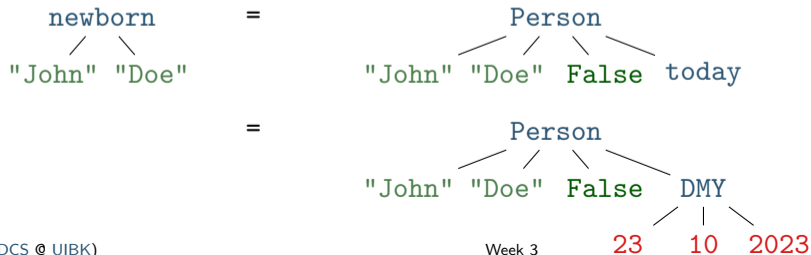
- task: define a function that takes first- and lastname and creates a (value of type) `Person` representing a newborn with that name

- solution:

```
today = DMY 23 10 2023
```

```
newborn fName lName = Person fName lName False today
```

- evaluation



Function Definitions using Patterns

- so far all functions definitions have been of the shape
`funName x1 ... xN = expr`
where `x1 ... xN` is a list of **variables**
- in these definitions we cannot inspect the structure of the input
- aim: define **functions depending on structure** of input
- example using vehicle datatype (with cars, bicycles and trucks)
 - task: convert a vehicle into a string
 - algorithm:
 - if the input is a **car with x PS**, then return "a car with x PS"
 - if the input is a **bicycle**, then return "a bicycle"
 - if the input is a **truck with x wheels**, then return "a(n) x -wheel truck"
- in Haskell, structure of trees are described by **patterns**
- the question whether some input tree fits a pattern is called **pattern matching**

Patterns

- a **pattern** is an expression of one of the following forms
 - `x` variable name as in a function definition
 - `_` underscore
 - `CName pat1 ... patN` constructor application with patterns `pat1 ... patN` as arguments
 - `x@pat` variable name followed by @ and pattern

where

- all variables occur at most once
 - numbers, strings, and characters can be interpreted as constructors
 - parentheses might be required for nested patterns
- examples

- `Car brand ps` ✓
- `Car _ ps` ✓
- `Car BMW 100` ✓
- `Car _ (50 + 50)` + is not a constructor ✗
- `Person "John" lName _ _` ✓
- `p@(Person _ _ _ (DMY 23 10 _))` ✓
- `Person name name _ _` duplicate variable ✗

Pattern Matching

- given an expression and a pattern, **pattern matching**
 - determines whether the expression is covered by the shape of the pattern,
 - and in the positive case determines a substitution of pattern-variables to expressions
- examples
 - `Car brand ps` matches `expr`,
if `expr` is an arbitrary car;
substitution contains both brand (in variable `brand`) and horsepower (in variable `ps`)
 - `Car _ ps` matches `expr`
if `expr` is an arbitrary car;
substitution will substitute `ps` by the horsepower, no interest in brand
 - `Car BMW 100` matches `expr`
if `expr` is a BMW with exactly 100 PS; substitution is empty
 - `Person "John" lName _ _` matches `expr`
if `expr` is a person whose first name is John; substitution contains last name in `lName`
 - `p@(Person _ _ _ (DMY 23 10 _))` matches `expr`
if `expr` is a person that can celebrate his/her birthdate today;
substitution will contain the full person in variable `p`

Pattern Matching Algorithm

- this slide contains an **algorithm for pattern matching**
- in the algorithm the substitution of variables to expressions is written as $x_1/\text{expr}_1, \dots, x_N/\text{expr}_N$
(here, / is not the division operator but the substitute operator)
- pattern matching algorithm for pattern **pat** and expression **expr**
 - **pat** is variable **x**: matching succeeds, substitution is x/expr
 - **pat** is **_**: matching succeeds, empty substitution
 - **pat** is $x@\text{pat}_1$: matching succeeds if **pat**₁ matches **expr**;
add x/expr to resulting substitution
 - **pat** is **CName pat**₁ ... **pat**_N:
 - if **expr** is **OtherCName** ... with **CName** \neq **OtherCName** then match fails
 - if **expr** is **CName expr**₁ ... **expr**_N then
match **expr**₁ with **pat**₁, ..., match **expr**_N with **pat**_N;
if all of these matches succeed then succeed with merged substitution, otherwise match fails
 - otherwise, first evaluate **expr** until outermost constructor is fixed
- remark: algorithm itself is described via pattern matching

Pattern Matching Algorithm – Examples

- try to match some patterns against expression `Car BMW (20 + 80)`
 - pattern `x`: success with substitution `x / Car BMW (20 + 80)`
 - pattern `Car brand ps`: success with substitution `brand / BMW, ps / (20 + 80)`
 - pattern `Car brand _`: success with substitution `brand / BMW`
 - pattern `Car Audi _`: failure
 - pattern `Car _ 100`: success with empty substitution, triggers evaluation
- next consider expression `Person "Liz" "Ball" True (DMY 23 10 1970)`
 - pattern `Person "John" lName _ _`: fails
 - pattern `p@(Person _ _ _ (DMY 23 10 _))`: success with substitution `p / Person "Liz" "Ball" True (DMY 23 10 1970)`

Function Definitions with Pattern Matching

- so far all functions definitions have been of shape

`funName x1 ... xN = expr`

- now add two generalizations
 - a function definition has the shape

`funName pat1 ... patN = expr`

(★)

where all variables in **patterns** `pat1 ... patN` occur at most once

- there can be **several equations** for the same function
- evaluation of `funName expr1 ... exprN` via function equation (★)
 - if `pat1` matches `expr1`, ..., `patN` matches `exprN` via some substitutions, then the equation is **applicable** and `funName expr1 ... exprN` is replaced by rhs `expr` with the merged substitution applied
 - otherwise, (★) is not applicable
- evaluation of `funName expr1 ... exprN`
 - apply first equation that is applicable (tried from top to bottom)
 - if no equation is applicable, abort computation with error

Function Definitions – Example on Person

```
data Date = DMY Int Int Integer
data Person = Person String String Bool Date
data Option = Some Integer | None
```

- task: change the last name of a person

```
withLastName lName (Person fName _ m b) = Person fName lName m b
```

remark: data is never changed but newly created

- task: compute the age of a person in years, if it is his or her birthday, otherwise return nothing

```
ageYear (Person _ _ _ (DMY 23 10 y)) = Some (2023 - y)
```

```
ageYear _ = None
```

remark: here the order of equations is important

- task: create a greeting for a person

```
greeting p@(Person name _ _ _) = gHelper name (ageYear p)
```

```
gHelper n None = "Hello " ++ n
```

```
gHelper n (Some a) = "Hi " ++ n ++ ", you turned " ++ show a
```

remark: (++) concatenates two strings, show converts values to strings

Merging Substitutions and Equality

- consider the following code for testing equality of two values

```
equal x x = True
equal _ _ = False
```
- consider evaluation of `equal 5 7`
 - first argument: `x` matches `5`, obtain substitution `x / 5`
 - second argument: `x` matches `7`, obtain substitution `x / 7`
 - merging these substitutions is not possible: `x / ???`
- Haskell avoids problem of non-mergeable substitutions by the distinct-variables-restriction in lhss, i.e., above definition is not allowed in Haskell
- correct solution for testing on equality
 - use `(==)`, a built-in operator to compares two values of the same type, the result will be of type `Bool`
 - for comparison of user-defined datatypes, replace `deriving Show` by `deriving (Show, Eq)`
 - examples: `5 == 7`, `"Peter" == name`, ..., but not `"five" == 5`

Function Definitions – Example on Bool

- consider built-in datatype `data Bool = True | False`

- consider function for conjunction of two Booleans

```
conj True b = b
```

```
conj False _ = False
```

- example evaluation (numbers are just used as index)

```
conj1 (conj2 True False) (conj3 True True)
```

```
-- check which equation is applicable for conj1
```

```
-- first equation triggers evaluation of first argument of conj1 (True)
```

```
-- check which equation is applicable for conj2
```

```
-- first equation is applicable with substitution b/False
```

```
= conj1 False (conj3 True True)
```

```
-- now see that only second equation is applicable for conj1
```

```
= False
```

- remark: many Boolean functions are predefined, e.g.,
(`&&`) (conjunction), (`||`) (disjunction),
(`/=`) (exclusive-or), `not` (negation)

Function Definitions by Case Analysis

- design principle for functions:
define equations to cover all possible shapes of input

- example

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
weekend Sat = True
```

```
weekend Sun = True
```

```
weekend _ = False
```

- example: first element of a list

```
data List = Empty | Cons Integer List
```

```
first (Cons x xs) = x
```

```
first Empty = error "first on empty list"
```

- `error` takes a string to deliver sensible error message upon evaluation
- without second defining equation, `first Empty` results in generic “non-exhaustive patterns” exception

Recursive Function Definitions

- example: length of a list

```
len Empty = 0
```

```
len (Cons x xs) = 1 + ??? -- the length of the list xs
```

- potential problem: we would like to apply a function that we are currently defining
- this is allowed in programming and called **recursion**:
a function definition that invokes itself

```
len Empty = 0
```

```
len (Cons x xs) = 1 + len xs -- len xs is recursive call
```

- make sure to have smaller arguments in recursive calls
- evaluation is as before

```
len (Cons 1 (Cons 7 (Cons 9 Empty)))  
= 1 + (len (Cons 7 (Cons 9 Empty)))  
= 1 + (1 + (len (Cons 9 Empty)))  
= 1 + (1 + (1 + (len Empty)))  
= 1 + (1 + (1 + 0)) = 1 + (1 + 1) = 1 + 2 = 3
```

Recursive Function Definitions – Example Append

- task: append two lists, e.g., appending [1, 5] and [3] yields [1, 5, 3]
- solution: pattern matching and recursion on first argument

```
append Empty ys = ys
```

```
append (Cons x xs) ys = Cons x (append xs ys)
```

- example evaluation

```
append (Cons 1 (Cons 3 Empty)) (Cons 2 (Cons 7 Empty))  
= Cons 1 (append (Cons 3 Empty) (Cons 2 (Cons 7 Empty)))  
= Cons 1 (Cons 3 (append Empty (Cons 2 (Cons 7 Empty))))  
= Cons 1 (Cons 3 (Cons 2 (Cons 7 Empty)))
```

Recursive Function Definitions – Evaluating Expr

- consider datatype for expressions

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr
```

- task: evaluate expression
- solution:

```
eval (Number x)    = x  
eval (Plus e1 e2)  = eval e1 + eval e2  
eval (Negate e)    = - eval e
```

Recursive Function Definitions – Expr to List

- consider datatype for expressions

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr
```

- task: create list of all numbers that occur in expression
- solution:

```
numbers (Number x)    = Cons x Empty  
numbers (Plus e1 e2) = append (numbers e1) (numbers e2)  
numbers (Negate e)    = numbers e
```

Summary

- function definitions by case analysis via **pattern matching**
 - patterns describe shapes of trees
 - multiple defining equations allowed, tried from top to bottom
- function definitions can be **recursive**
 - `funName ... = ... (funName ...) ... (funName ...) ...`
 - arguments in recursive call should be smaller than in lhs